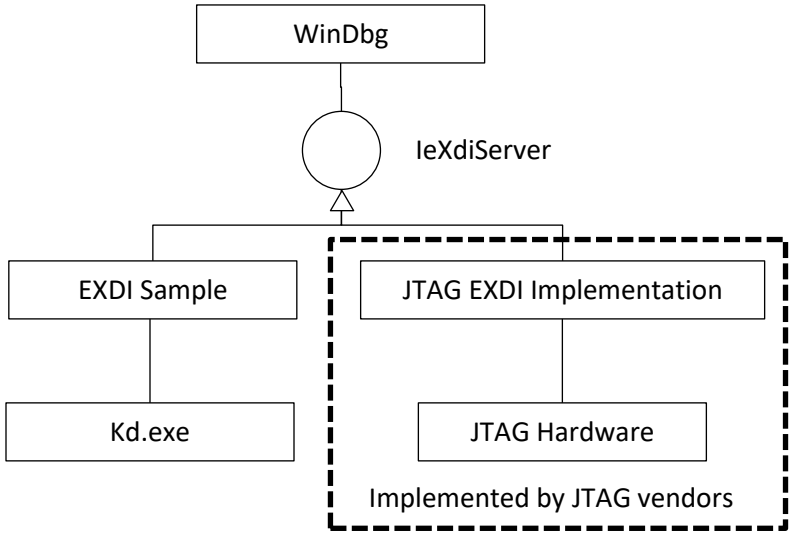
EXDI KD Sample – getting started

EXDI is an interface that allows extending WinDbg by adding support for hardware debuggers (e.g. JTAG-based). This sample is intended for JTAG debugger vendors that want to add support for their hardware to WinDbg and other Microsoft debuggers. The diagram below illustrates the role of EXDI:



The sample consists of 2 parts:

- A "static" sample. It supports viewing the state of a stopped target (e.g. view stack/variables/modules) and does not support resuming or setting breakpoints.
- A "live" sample. Additionally to all functionality of the static sample it supports stepping, setting breakpoints and resuming target execution.

Static EXDI sample

In order to make WinDbg support a third-party JTAG debugger in the "static mode" (analyze the state of a stopped target, no support for resuming or setting breakpoints) the JTAG vendor needs to provide an implementation of the EXDI interface supporting methods that:

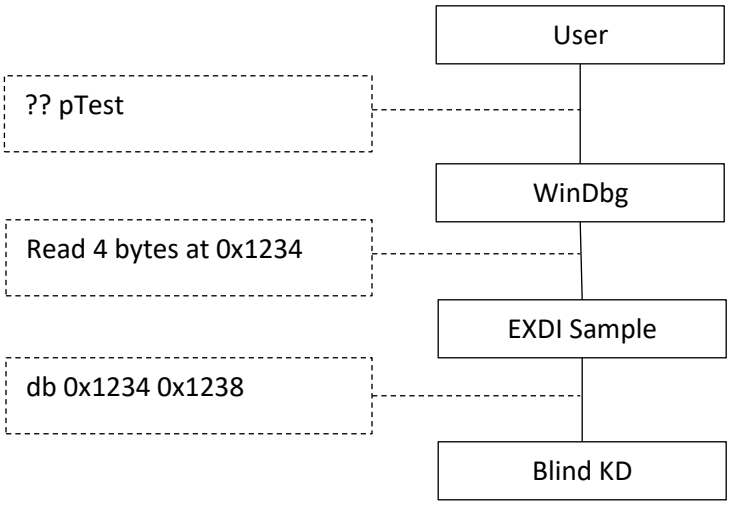
- Read virtual memory at a given address
- Read CPU registers

This is enough to provide debugging experience similar to analyzing crash dumps – WinDbg will handle symbols, unwind stacks and parse OS-specific structures.

This example does not depend on any real hardware. Instead we "emulate" a JTAG device by running command-line kernel debugger (kd.exe) and artificially restricting it to 2 basic commands:

- The 'db' command to read memory
- The 'r' command to read registers

We refer to the restricted kd.exe as “blind KD”. The example demonstrates how higher-level commands (e.g. evaluating a C++ expression) will be translated by Microsoft debugger engine into series of low-level commands, such as ‘read memory’ and handled by the EXDI implementation:



JTAG vendors should implement those basic operations using their JTAG programmers using this example as a reference.

This document assumes that the reader has basic experience debugging Windows drivers. If not, please refer to MSDN and WDK documentation for instructions on building and debugging a basic driver. It is recommended to deploy the driver on a virtual machine (e.g. Hyper-V).

Before you begin

Before starting to do anything with this example, please follow the preparation steps below:

1. Install Debugging Tools for Windows. It is recommended to use the 32-bit version of the tools.
2. Build the ExdiKdSample.sln solution and register ExdiKdSample.dll produced by the build by running ‘regsvr32 ExdiKdSample.dll’ as Administrator.
3. Create a driver project containing the following code:

```
#include <wdm.h>

extern "C" NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject,
PUNICODE_STRING RegistryPath)
{
    (void)DriverObject, (void)RegistryPath;

    const char *pTest = "Hello, World\n";
    DbgPrint("%s", pTest);
    DbgBreakPoint();
    return STATUS_NOT_IMPLEMENTED;
}
```

4. Build the driver, deploy it on a second machine (e.g. a Hyper-V virtual machine) and ensure that you can debug it with WinDbg. Note the command-line arguments used to launch WinDbg (e.g. `-k com:pipe,port=\\.\pipe\vmkerneltest1`).

5. Take a note of the **KdVersionBlock** address in the kernel you are about to debug. Connect to the kernel using normal WinDbg, break-in and run the following command:

```
kd> ? KdVersionBlock
Evaluate expression: -8788337193488 = fffff801`ce488df0
```

You will need the underlined decimal value later. Decimal is used for compatibility reasons.

Analyzing the system state with EXDI

We will now start a normal kernel debugging session with WinDbg, stop the kernel at a certain point, disconnect WinDbg and reconnect using EXDI. This document will explain how work is split between WinDbg (handling symbols) and the EXDI Server (fetching memory/registers). Please follow the steps below to get started:

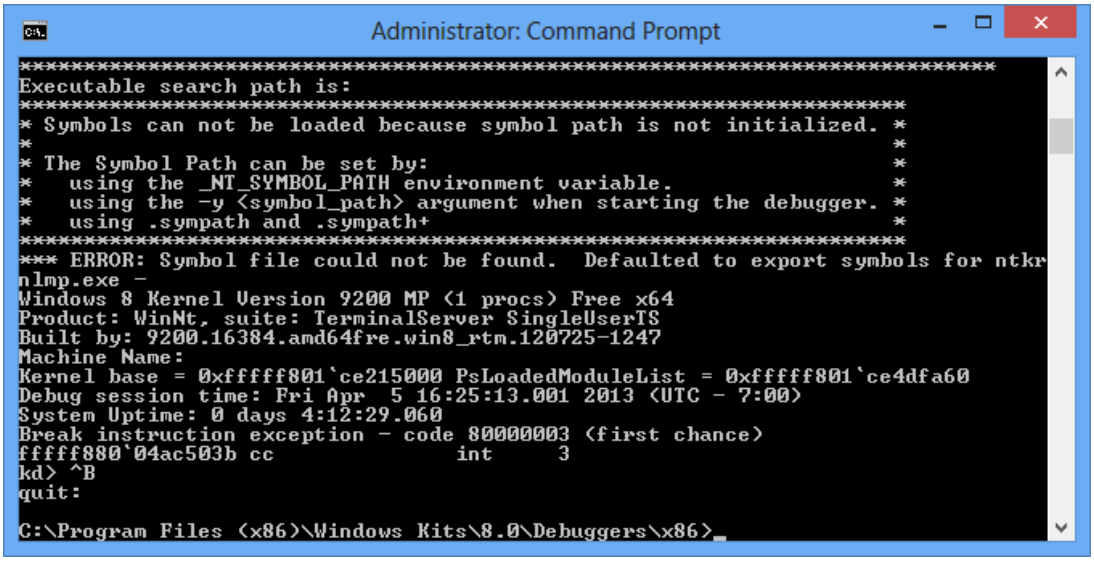
1. Start a normal kernel debugging session with WinDbg. Load the driver and wait until it stops on the **DbgBreakPoint()** call.
2. Ensure you have recorded the **KdVersionBlock** address.
3. Open a new Command Prompt window in Administrator mode and go to the Debugging Tools directory.
4. Try running kd.exe manually. E.g. when debugging an ARM tablet over USB having 'surface' as the debug target name, run the following command:

```
kd -k usb:targetname=surface
```

If you are debugging a virtual machine with COM1 redirected to a pipe called 'vmpipe', run:

```
kd -k com:pipe,reconnect,port=\\.\pipe\vmpipe
```

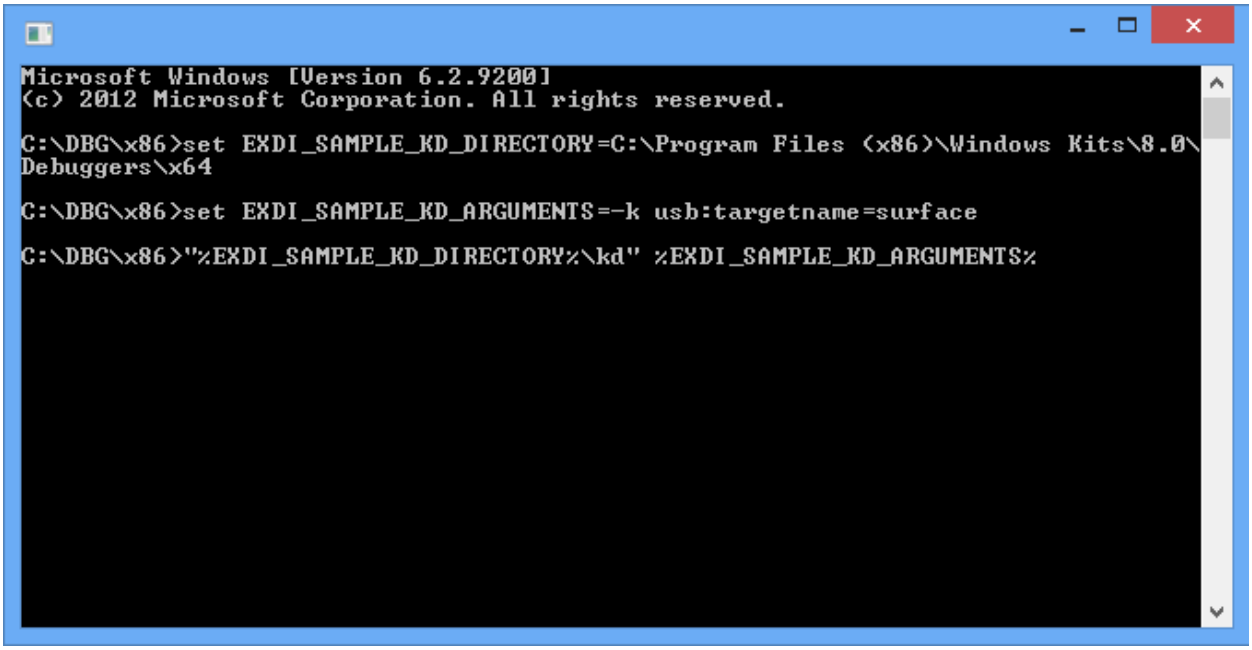
5. Ensure that KD can connect to the kernel. Exit it by pressing Ctrl-B followed by ENTER. **Do not use the 'q' command, as it would resume the kernel.**



6. Set the following environment variables:
    a. EXDI_SAMPLE_KD_DIRECTORY to the directory containing kd.exe (normally same directory as the current one).
    b. EXDI_SAMPLE_KD_ARGUMENTS to the arguments used to start kd.exe.

7. Recheck your arguments by running the following command from the command window with the environment variables:

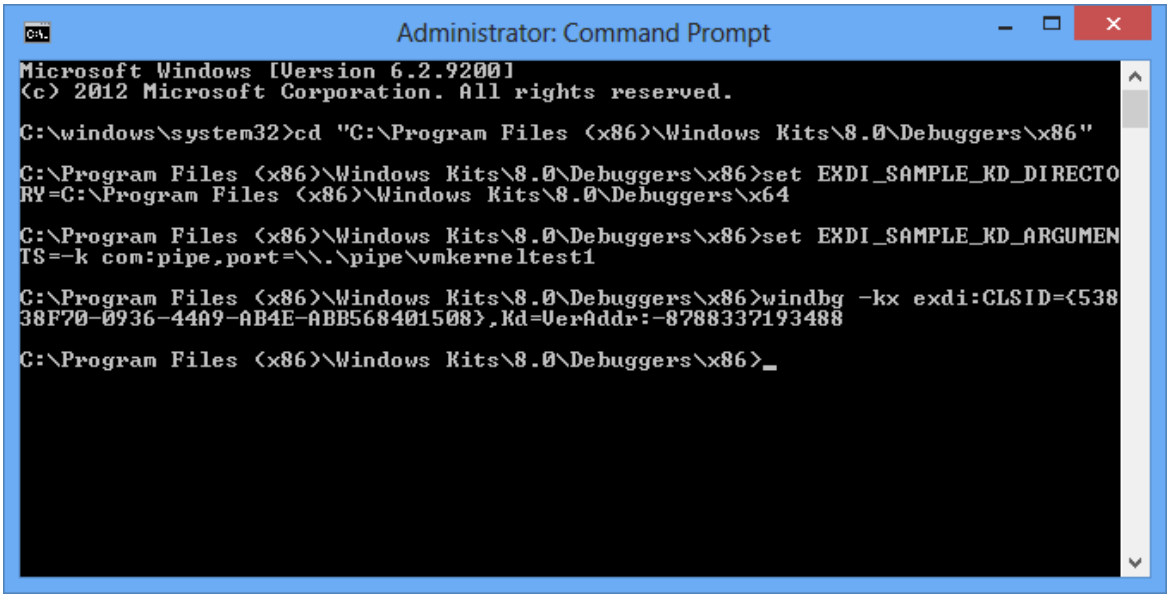
```
“%EXDI_SAMPLE_KD_DIRECTORY%\kd” %EXDI_SAMPLE_KD_ARGUMENTS%
```



If KD starts successfully, exit it by pressing Ctrl-B followed by Enter.

8. Run WinDbg with the following arguments from the same command prompt:

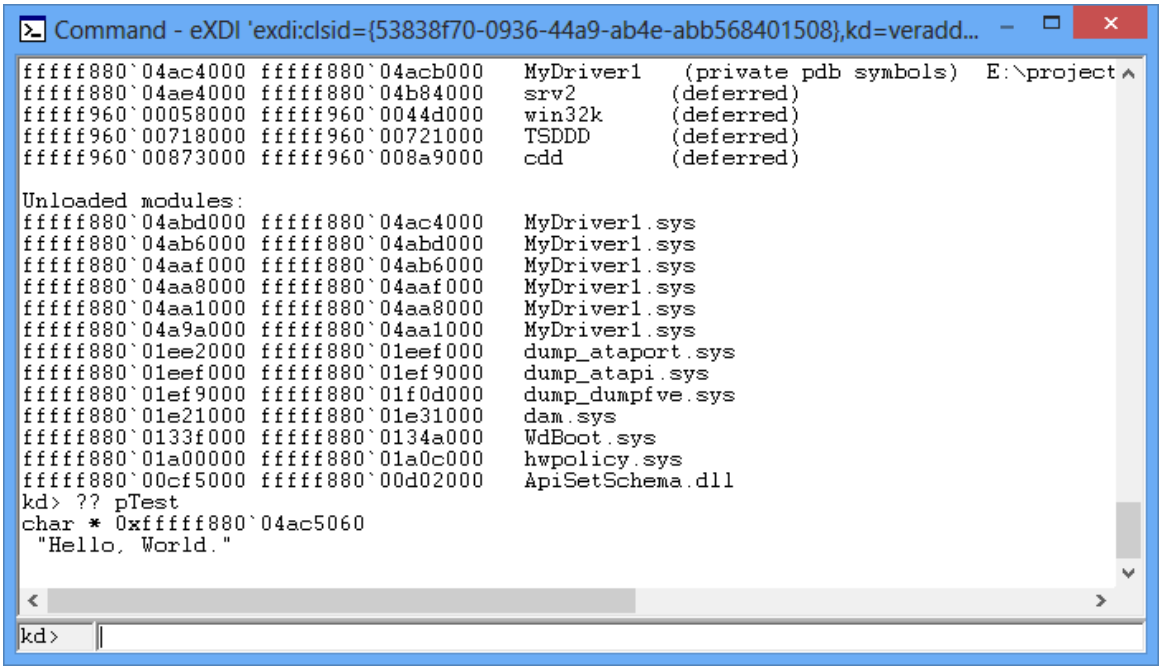
```
-kx exdi:CLSID={53838F70-0936-44A9-AB4E-ABB568401508},Kd=VerAddr:<Address of KdVersionBlock in decimal>
```



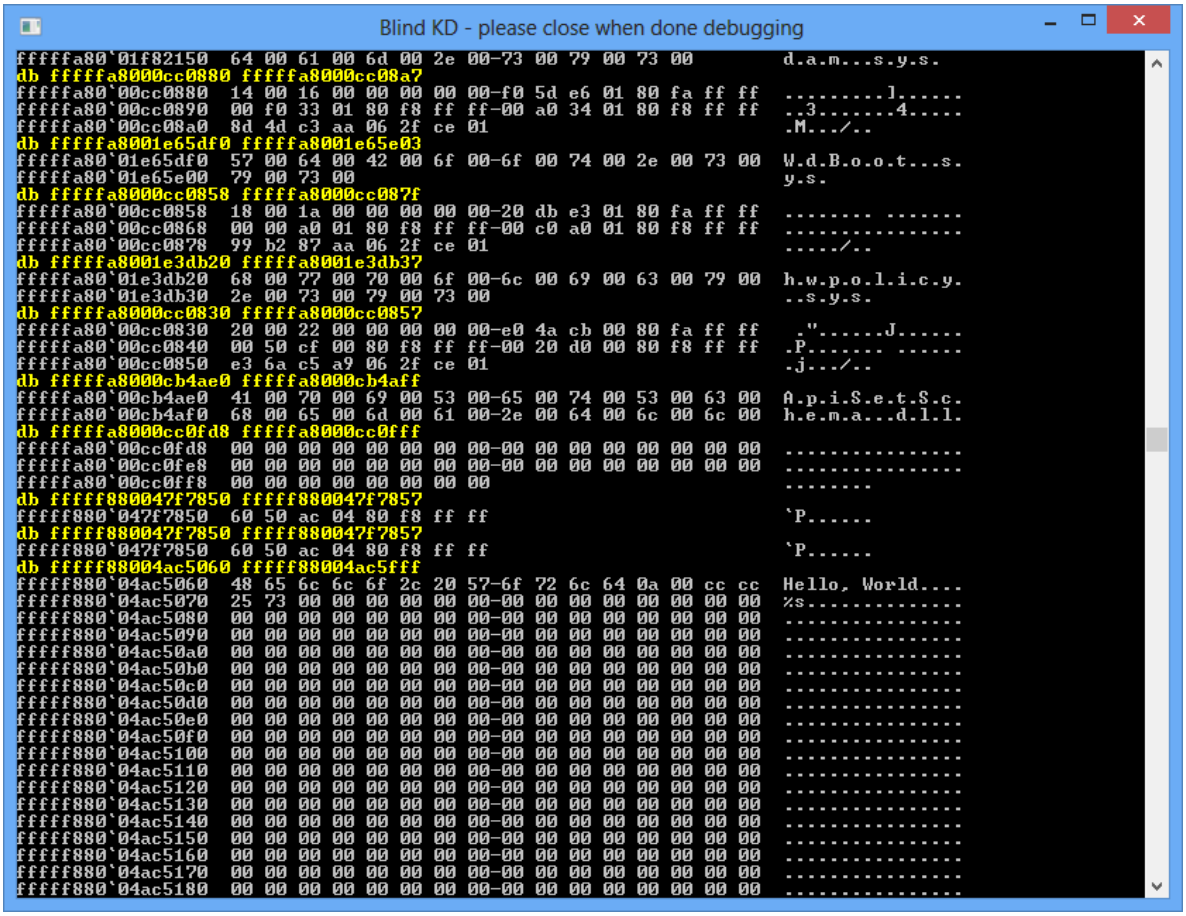
9. You will see the normal WinDbg window, a kd.exe window and the ‘Blind KD Output’ window. Go to the normal WinDbg window and ensure that the symbols are loaded:

```
.symfix c:\symbols.net
.reload
```

10. Position the windows so that you can see the ‘Blind KD’ window while interacting with WinDbg.
11. WinDbg will show that the kernel is stopped at the DbgBreakPoint() call. Run the ‘?? pTest’ command or hover the mouse over pTest to see its value:



12. Note the output in the ‘Blind KD’ window:



When you entered the '?? pTest' command, WinDbg used the PDB symbols and the loaded module structures in the Windows kernel to compute the address of the pTest variable (**0xfffff880047f7850** in this example). It then asked our EXDI server to read 8 bytes at that memory location. Then WinDbg interpreted those bytes as a pointer and read the rest of the page containing the string to show it to the user in a more informative way. The EXDI server did not participate in any symbol-related activities – it simply fetched the raw memory contents requested by WinDbg and WinDbg interpreted it.

13. You can run other WinDbg commands and observe how they are translated into memory reading commands handled by the EXDI server. When done, exit WinDbg, and forcibly close the 'Blind KD' and KD.EXE windows.

Live EXDI sample

Once you finished trying out the static EXDI server, replace the CLSID in the WinDbg command line to {67030926-1754-4FDA-9788-7F731CBDAE42}. This will activate a more advanced sample server that supports running the target, setting breakpoints and stepping through the code.

Developing your own EXDI server

In order to add support for your JTAG hardware to WinDbg, all you need to do is create an EXDI server implementing basic operations such as memory reading.

It is recommended to start with modifying the static EXDI sample (`CStaticExdiSampleServer` class). To get minimum functionality you will need to change the following methods:

- `CStaticExdiSampleServer::ReadVirtualMemory()`
- `CStaticExdiSampleServer::GetContext()`
- `CStaticExdiSampleServer::GetRunStatus()`